Programming Assignment 1: Percolation

Write a program to estimate the value of the *percolation threshold* via Monte Carlo simulation.

**Install a Java programming environment.** Install a Java programming environment on your computer by following these step-by-step instructions for your operating system [ [Mac OS X](http://algs4.cs.princeton.edu/mac) · [Windows](http://algs4.cs.princeton.edu/windows) · [Linux](http://algs4.cs.princeton.edu/linux) ]. After following these instructions, the commands javac-algs4 and java-algs4 will classpath in [algs4.jar](http://algs4.cs.princeton.edu/code/algs4.jar), which contains Java classes for I/O and all of the algorithms in the textbook.

*Note that, as of August 2015, you must use the named package version of algs.jar. To access a class, you need an import statement, such as the ones below:*

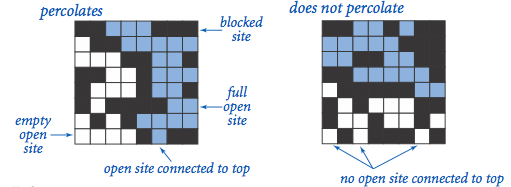
import edu.princeton.cs.algs4.StdRandom;

import edu.princeton.cs.algs4.StdStats;

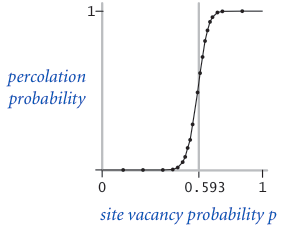
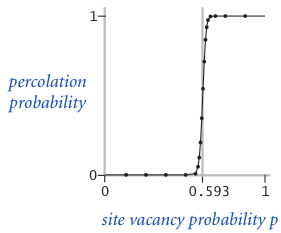
import edu.princeton.cs.algs4.WeightedQuickUnionUF;

**Percolation.** Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as *percolation* to model such situations.

**The model.** We model a percolation system using an *N*-by-*N* grid of *sites*. Each site is either *open* or *blocked*. A *full* site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system *percolates* if there is a full site in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. (For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.)



**The problem.** In a famous scientific problem, researchers are interested in the following question: if sites are independently set to be open with probability *p* (and therefore blocked with probability 1 − *p*), what is the probability that the system percolates? When *p* equals 0, the system does not percolate; when *p* equals 1, the system percolates. The plots below show the site vacancy probability *p* versus the percolation probability for 20-by-20 random grid (left) and 100-by-100 random grid (right).

When *N* is sufficiently large, there is a *threshold* value *p*\* such that when *p* < *p*\* a random *N*-by-*N* grid almost never percolates, and when *p* > *p*\*, a random *N*-by-*N* grid almost always percolates. No mathematical solution for determining the percolation threshold *p*\* has yet been derived. Your task is to write a computer program to estimate *p*\*.

**Percolation data type.** To model a percolation system, create a data type Percolation with the following API:

**public class Percolation {**

**public Percolation(int N)** // create N-by-N grid, with all sites blocked

**public void open(int i, int j)** // open site (row i, column j) if it is not open already

**public boolean isOpen(int i, int j)** // is site (row i, column j) open?

**public boolean isFull(int i, int j)** // is site (row i, column j) full?

**public boolean percolates()** // does the system percolate?

**public static void main(String[] args)** // test client (optional)

**}**

*Corner cases.* By convention, the row and column indices *i* and *j* are integers between 1 and *N*, where (1, 1) is the upper-left site: Throw a java.lang.IndexOutOfBoundsException if any argument to open(),isOpen(), or isFull() is outside its prescribed range. The constructor should throw a java.lang.IllegalArgumentException if *N* ≤ 0.

*Performance requirements.* The constructor should take time proportional to *N*2; all methods should take constant time plus a constant number of calls to the union-find methods union(), find(), connected(), andcount().

**Monte Carlo simulation.** To estimate the percolation threshold, consider the following computational experiment:

* Initialize all sites to be blocked.
* Repeat the following until the system percolates:
  + Choose a site (row *i*, column *j*) uniformly at random among all blocked sites.
  + Open the site (row *i*, column *j*).
* The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20-by-20 lattice according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Percolation 50 sites  *50 open sites* | Percolation 100 sites  *100 open sites* | Percolation 150 sites  *150 open sites* | Percolation 204 sites  *204 open sites* |

By repeating this computation experiment *T* times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let *xt* be the fraction of open sites in computational experiment *t*. The sample mean μ provides an estimate of the percolation threshold; the sample standard deviation σ measures the sharpness of the threshold.

Estimating the sample mean and variance

Assuming *T* is sufficiently large (say, at least 30), the following provides a 95% confidence interval for the percolation threshold:

95% confidence interval for percolation threshold

To perform a series of computational experiments, create a data type PercolationStats with the following API.

**public class PercolationStats {**

**public PercolationStats(int N, int T)**  // perform T independent experiments on an N-by-N grid

**public double mean()**  // sample mean of percolation threshold

**public double stddev()**  // sample standard deviation of percolation threshold

**public double confidenceLo()**  // low endpoint of 95% confidence interval

**public double confidenceHi()**  // high endpoint of 95% confidence interval

**public static void main(String[] args)**  // test client (described below)

**}**

The constructor should throw a java.lang.IllegalArgumentException if either *N* ≤ 0 or *T* ≤ 0.

Also, include a main() method that takes two *command-line arguments* *N* and *T*, performs *T* independent computational experiments (discussed above) on an *N*-by-*N* grid, and prints the mean, standard deviation, and the *95% confidence interval* for the percolation threshold. Use [StdRandom](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdRandom.html) to generate random numbers; use [StdStats](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdStats.html) to compute the sample mean and standard deviation.

% **java PercolationStats 200 100**

mean = 0.5929934999999997

stddev = 0.00876990421552567

95% confidence interval = 0.5912745987737567, 0.5947124012262428

% **java PercolationStats 200 100**

mean = 0.592877

stddev = 0.009990523717073799

95% confidence interval = 0.5909188573514536, 0.5948351426485464

% **java PercolationStats 2 10000**

mean = 0.666925

stddev = 0.11776536521033558

95% confidence interval = 0.6646167988418774, 0.6692332011581226

% **java PercolationStats 2 100000**

mean = 0.6669475

stddev = 0.11775205263262094

95% confidence interval = 0.666217665216461, 0.6676773347835391

**Analysis of running time and memory usage (optional and not graded).** Implement the Percolation data type using the *quick find* algorithm in [QuickFindUF](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/QuickFindUF.html).

* Use [Stopwatch](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/Stopwatch.html) to measure the total running time of PercolationStats for various values of N and T. How does doubling *N* affect the total running time? How does doubling *T* affect the total running time? Give a formula (using tilde notation) of the total running time on your computer (in seconds) as a single function of both *N* and *T*.
* Using the 64-bit memory-cost model from lecture, give the total memory usage in bytes (using tilde notation) that a Percolation object uses to model an *N*-by-*N* percolation system. Count all memory that is used, including memory for the union-find data structure.

Now, implement the Percolation data type using the *weighted quick union* algorithm in [WeightedQuickUnionUF](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/WeightedQuickUnionUF.html). Answer the questions in the previous paragraph.

**Deliverables.** Submit only Percolation.java (using the weighted quick-union algorithm from [WeightedQuickUnionUF](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/WeightedQuickUnionUF.html)) and PercolationStats.java. We will supply algs4.jar. Your submission may not call library functions except those in [StdIn](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdIn.html), [StdOut](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdOut.html), [StdRandom](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdRandom.html), [StdStats](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdStats.html), [WeightedQuickUnionUF](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/WeightedQuickUnionUF.html), and java.lang.

**For fun.** Create your own percolation input file and share it in the discussion forums. For some inspiration, see these [nonogram puzzles](http://photo.qip.ru/users/syusyu/2208847/?sort=visitors&page=1).

Programming Assignment 1 Checklist: Percolation

|  |
| --- |
| **Frequently Asked Questions (General)** |

**What's a checklist?** The assignment provides the programming assignment specification; the checklist provides clarifications, test data, and hints that might be helpful in completing the assignment.

**Which Java programming environment should I use?** For novices, we recommend the lightweight IDE [DrJava](http://www.drjava.org/) along with the command line. If you use our Mac OS X or Windows installer, then everything should be configured and ready to go. If you prefer to use a different IDE (such as Eclipse), that's perfectly fine too—just be sure that you know how to do the following:

* Add algs4.jar to your Java classpath.
* Enter command-line arguments.
* Use standard input and standard output (and, ideally, redirect them to or from a file).

**Where can I find the Java code and Javadoc for the algorithms and data structures from lecture and the textbook?** They are in algs4.jar. Here are the [APIs](http://algs4.cs.princeton.edu/code/index.php).

**Where can I find the Javadoc for the input and output libraries from lecture and the textbook?** They are in algs4.jar. Here are the [APIs](http://algs4.cs.princeton.edu/code/index.php#stdlib).

**How can I classpath in the textbook libraries from the command line?** If you use our Mac OS X or Windows installer, then you can automatically classpath in the textbook libraries using the commands javac-algs4and java-algs4 (instead of javac and java).

**I haven't programmed in Java in a while. What material do I need to remember?** For a review of our Java programming model (including our input and output libraries), read Sections 1.1 and 1.2 of Algorithms, 4th Edition.

**Can I use various Java libraries in this assignment, such as java.util.LinkedList, java.util.ArrayList, java.util.TreeMap, and java.util.HashMap?** No. You should not use any Java libraries until we have implemented equivalent versions in lecture. Once we have introduced them in lecture, you are free to use either the Java library version or our equivalent. You are welcome to use classes in the Java language such as Math.sqrt() and Integer.parseInt().

**How do I throw a java.lang.IndexOutOfBoundsException?** Use a throw statement like the following:

if (i <= 0 || i > N) throw new IndexOutOfBoundsException("row index i out of bounds");

Your code should not attempt to catch any exceptions—this will interfere with our grading scripts.

**How should I format and comment my code?** Here are some recommended [style guidelines](http://introcs.cs.princeton.edu/11style/). Below are some that are particularly important (though we will not deduct for style in this course).

* Include a bold (or Javadoc) comment at the beginning of each file with your name, date, the purpose of the program, and how to execute it.
* Include a bold (or Javadoc) comment describing every method.
* Include a comment describing every instance variable.
* Indent consistently, using 3 or 4 spaces for each indentation level. Do not use hard tabs.
* Do not exceed 80 characters per line. This rule also applies to the readme.txt file.
* Avoid unexplained magic numbers, especially ones that are used more than once.

|  |
| --- |
| **Frequently Asked Questions (Percolation)** |

**What are the goals of this assignment?**

* Set up a Java programming environment.
* Use our input and output libraries.
* Learn about a scientific application of the union–find data structure.
* Measure the running time of a program and use the doubling hypothesis to make predictions.
* Measure the amount of memory used by a data structure.

**Can I add (or remove) methods to (or from) Percolation?** No. You must implement the Percolation API exactly as specified, with the identical set of public methods and signatures or your assignment will not be graded. However, you are encouraged to add private methods that enhance the readability, maintainability, and modularity of your program. The one exception is main()—you are always permitted to add this method to test your code, but we will not call it unless we specify it in our API.

**Can my Percolation data type assume the row and column indices are between 0 and N−1?** No. The API specifies that valid row and column indices are between 1 and N.

**Why is it so important to implement the prescribed API?** Writing to an API is an important skill to master because it is an essential component of modular programming, whether you are developing software by yourself or as part of a group. When you develop a module that properly implements an API, anyone using that module (including yourself, perhaps at some later time) does not need to revisit the details of the code for that module when using it. This approach greatly simplifies writing large programs, developing software as part of a group, or developing software for use by others.

Most important, when you properly implement an API, others can write software to use your module or to test it. We do this regularly when grading your programs. For example, your PercolationStats client should work with our Percolation data type and vice versa. If you add an extra public method to Percolation and call them from PercolationStats, then your client won't work with our Percolation data type. Conversely, our PercolationStats client may not work with your Percolation data type if you remove a public method.

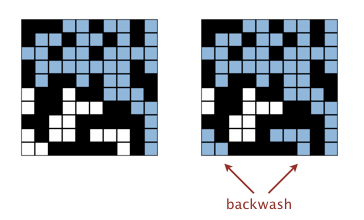
**How many lines of code should my program be?** You should strive for clarity and efficiency. Our reference solution for Percolation.java is about 70 lines, plus a test client. Our PercolationStats.java client is about 50 lines. If you are re-implementing the union-find data structure (instead of reusing the implementations provided), you are on the wrong track.

**What assumptions can I make about the input to main() in PercolationStats?** It can be any valid input: an integer N ≥ 1 and an integer T ≥ 1. In general, in this course you can assume that the input is of the specified format. But you do need to deal with pathological cases such as N = 1.

**What should stddev() return if T equals 1?** The sample standard deviation is undefined. We recommend returning Double.NaN.

**After the system has percolated, my PercolationVisualizer colors in light blue all sites connected to open sites on the bottom (in addition to those connected to open sites on the top). Is this "backwash" acceptable?** No, this is likely a bug in Percolation. It is only a minor deduction (because it impacts only the visualizer and not the experiment to estimate the percolation threshold), so don't go crazy trying to get this detail. However, many students consider this to be the most challenging and creative part of the assignment (especially if you limit yourself to one union-find object).

   % **java PercolationVisualizer input10.txt**



**How do I generate a site uniformly at random among all blocked sites for use in PercolationStats?** Pick a site at random (by using StdRandom to generate two integers between 1 and N) and use this site if it is blocked; if not, repeat.

**I don't get reliable timing information in PercolationStats when N = 200. What should I do?** Increase the size of N (say to 400, 800, and 1600), until the mean running time exceeds its standard deviation.

|  |
| --- |
| **Style and Bug Checkers** |

**Style checker.** We recommend using [Checkstyle 6.9](http://checkstyle.sourceforge.net/) (and the configuration file [checkstyle.xml](http://algs4.cs.princeton.edu/mac/checkstyle.xml)) to check the style of your Java programs. Here is a list of available [Checkstyle checks](http://checkstyle.sourceforge.net/availablechecks.html).

**Bug checker.** We recommend using [FindBugs 2.0.3](http://findbugs.sourceforge.net/) (and the configuration file [findbugs.xml](http://algs4.cs.princeton.edu/mac/findbugs.xml)) to identify common bug patterns in your code. Here is a summary of [FindBugs Bug descriptions](http://findbugs.sourceforge.net/bugDescriptions.html).

**Mac OS X and Windows installer.** If you used our Mac OS X or Windows installer, these programs are already installed as command-line utilities. You can check a single file or multiple files via the commands:

% **checkstyle-algs4 HelloWorld.java**

% **checkstyle-algs4 \*.java**

% **findbugs-algs4 HelloWorld.class**

% **findbugs-algs4 \*.class**

Note that Checkstyle inspects the source code; Findbugs inspects the compiled code.

**Eclipse.** For Eclipse users, there is a [Checkstyle plugin for Eclipse](http://eclipse-cs.sourceforge.net/) and a [Findbugs plugin for Eclipse](http://findbugs.cs.umd.edu/eclipse).

**Caveat.** The appearance of a warning message does not necessarily lead to a deduction (and, in some cases, it does not even indicate an error).

|  |
| --- |
| **Testing** |

**Testing.** We provide two clients that serve as large-scale visual traces. We highly recommend using them for testing and debugging your Percolation implementation.

**Visualization client.** [PercolationVisualizer.java](http://coursera.cs.princeton.edu/algs4/testing/percolation/PercolationVisualizer.java) animates the results of opening sites in a percolation system specified by a file by performing the following steps:

* Read the grid size N from the file.
* Create an N-by-N grid of sites (initially all blocked).
* Read in a sequence of sites (row i, column j) to open from the file. After each site is opened, draw full sites in light blue, open sites (that aren't full) in white, and blocked sites in black using standard draw, with with site (1, 1) in the upper left-hand corner.

The program should behave as in [this movie](http://coursera.cs.princeton.edu/algs4/checklists/percolation20-by-20.mov) and the following snapshots when used with [input20.txt](http://coursera.cs.princeton.edu/algs4/testing/percolation/input20.txt).

% **java PercolationVisualizer input20.txt**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Percolation 50 sites  50 open sites | Percolation 100 sites  100 open sites | Percolation 150 sites  150 open sites | Percolation 204 sites  204 open sites | Percolation 250 sites  250 open sites |

**Sample data files.** The directory [percolation](http://coursera.cs.princeton.edu/algs4/testing/percolation/) contains some sample files for use with the visualization client. Associated with each input .txt file is an output .png file that contains the desired graphical output at the end of the animation. For convenience, [percolation-testing.zip](http://coursera.cs.princeton.edu/algs4/testing/percolation-testing.zip) contains all of these files bundled together.

**InteractiveVisualization client.** [InteractivePercolationVisualizer.java](http://coursera.cs.princeton.edu/algs4/testing/percolation/InteractivePercolationVisualizer.java) is similar to the first test client except that the input comes from a mouse (instead of from a file). It takes a command-line integer N that specifies the lattice size. As a bonus, it writes to standard output the sequence of sites opened in the same format used by PercolationVisualizer, so you can use it to prepare interesting files for testing. If you design an interesting data file, feel free to share it with us and your classmates by posting it in the discussion forums.

|  |
| --- |
| **Possible Progress Steps** |

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Consider not worrying about backwash for your first attempt.** If you're feeling overwhelmed, don't worry about backwash when following the possible progress steps below. You can revise your implementation once you have a better handle on the problem and have solved the problem without handling backwash.
2. **For each method in Percolation that you must implement (open(), percolates(), etc.), make a list of which WeightedQuickUnionUF methods might be useful for implementing that method.** This should help solidify what you're attempting to accomplish.
3. **Using the list of methods above as a guide, choose instance variables that you'll need to solve the problem.** Don't overthink this, you can always change them later. Instead, use your list of instance variables to guide your thinking as you follow the steps below, and make changes to your instance variables as you go. Hint: At minimum, you'll need to store the grid size, which sites are open, and which sites are connected to which other sites. The last of these is exactly what the union-find data structure is designed for.
4. **Plan how you're going to map from a 2-dimensional (row, column) pair to a 1-dimensional union find object index.** You will need to come up with a scheme for uniquely mapping 2D coordinates to 1D coordinates. We recommend writing a private method with a signature along the lines of int xyTo1D(int, int) that performs this conversion. You will need to utilize the percolation grid size when writing this method. Writing such a private method (instead of copying and pasting a conversion formula multiple times throughout your code) will greatly improve the readability and maintainability of your code. In general, we encourage you to write such modules wherever possible. Directly test this method using the main() function of Percolation.
5. **Write a private method for validating indices.** Since each method is supposed to throw an exception for invalid indices, you should write a private method which performs this validation process.
6. **Write the open() method and the Percolation() constructor.** The open() method should do three things. First, it should validate the indices of the site that it receives. Second, it should somehow mark the site as open. Third, it should perform some sequence of WeightedQuickUnionUF operations that links the site in question to its open neighbors. The constructor and instance variables should facilitate the open()method's ability to do its job.
7. **Test the open() method and the Percolation() constructor.** These tests should be in main(). An example of a simple test is to call open(1, 1) and open(1, 2), and then to ensure that the two corresponding entries are connected (using .connected() in WeightedQuickUnionUF).
8. **Write the percolates(), isOpen(), and isFull() methods.** These should be very simple methods.
9. **Test your complete implementation using the visualization clients.**
10. **Write and test the PercolationStats class.**

|  |
| --- |
| **Programming Tricks and Common Pitfalls** |

1. **Do not write your own union–find data structure. Use WeightedQuickUnionUF instead.**
2. **Your Percolation class must use**[**WeightedQuickUnionUF**](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/WeightedQuickUnionUF.html)**.** Otherwise, it will fail the timing tests, as the autograder intercepts and counts calls to methods in WeightedQuickUnionUF.
3. **It's OK to use an extra row and/or column to deal with the 1-based indexing of the percolation grid.** Though it is slightly inefficient, it's fine to use arrays or union-find objects that are slightly larger than strictly necessary. Doing this results in cleaner code at the cost of slightly greater memory usage.
4. **Each of the methods (except the constructor) in Percolation must use a constant number of union-find operations.** If you have a for-loop inside of one of your Percolation methods, you're probably doing it wrong. Don't forget about the virtual-top / virtual-bottom trick described in lecture.